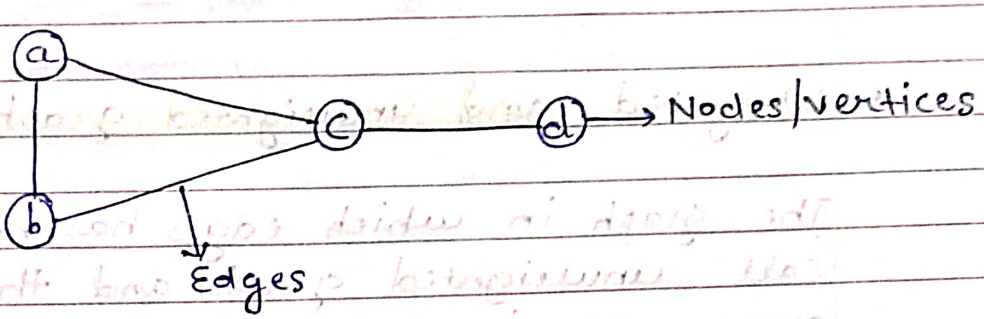
24/01/2024
Wednesday

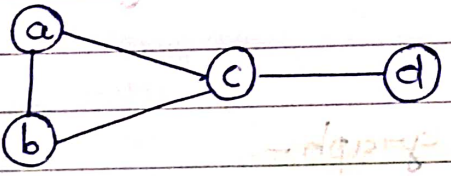
* **Graph**

Graph is a data structure comprises nodes and edges.

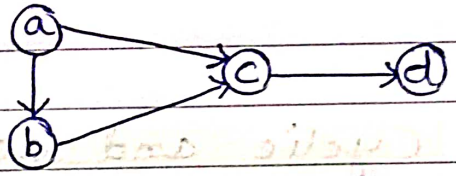


= Edges → directed
→ un-directed

undirected



directed



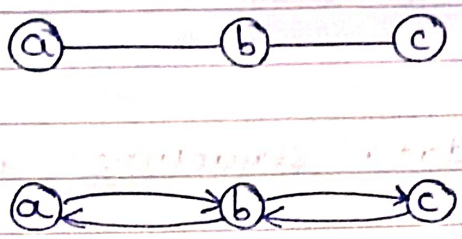
Edge list -

a → b	b → a
a → c	c → a
b → c	c → b
c → d	d → c

Edge list -

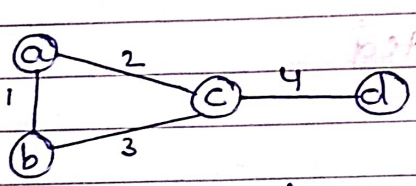
a → b
a → c
b → c
c → d

- **Converting undirected graph into directed –**

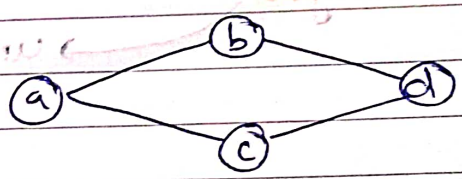


- **Weighted and unweighted graph –**

The graph in which edges has no weight is call unweighted graph and the graph in which every edge is assigned a weight is called weighted graph.



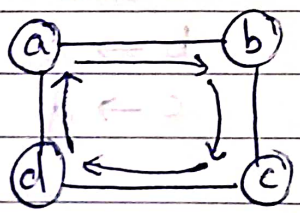
weighted graph



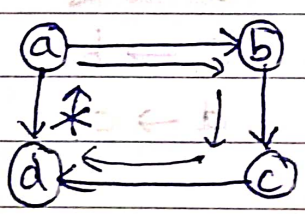
Unweighted graph

- **Cyclic and acyclic graph –**

If we traverse graph such that beginning and ending node is same if possible, then graph is cyclic otherwise acyclic.



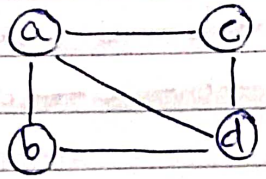
Cyclic graph



Acyclic graph

- **Degree of a node in graph-**

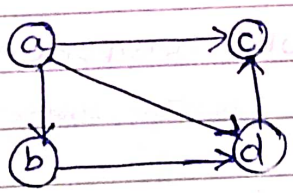
Degree is the number of edges connected to a node.



degree (a) = 3
degree (b) = 2
degree (c) = 2
degree (d) = 3

- **Indegree and outdegree-**

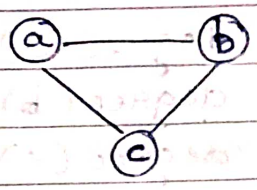
Total incoming edges to a node is indegree.
Total outgoing edges to a node is outdegree.



indegree (a) = 0
outdegree (a) = 3
indegree (b) = 1
outdegree (b) = 1
indegree (c) = 2
outdegree (c) = 0
indegree (d) = 2
outdegree (d) = 1

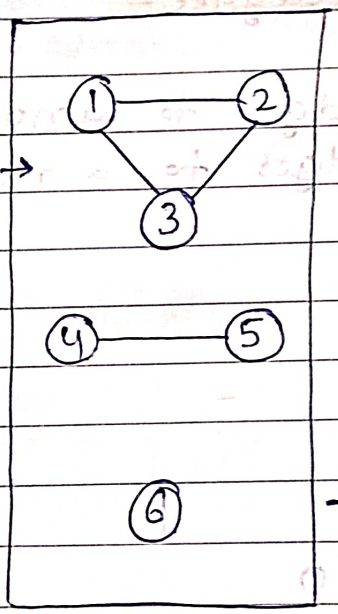
- **Path in a graph-**

Is a sequence of nodes in which no node is repeated.



a—b—c (valid path)
a—b—c—a (invalid path)

- **Components of graph-**



Disconnected graph →

First component

Second component

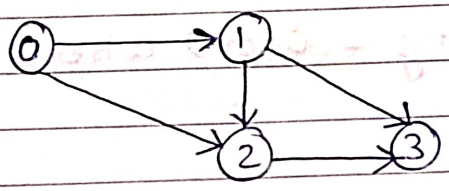
Third component

* **Graph creation-**

- **Adjacency Matrix** - It's a way of representing a graph as a matrix of boolean (0's and 1's).

For n nodes in graph, the dimension of 2-D matrix will be nxn.
If there is an edge from node i to J, mark adjMatrix[i][J] = 1, otherwise 0.

let the graph is -



Edge list = $0 \rightarrow 1$ $1 \rightarrow 3$
$0 \rightarrow 2$ $2 \rightarrow 3$
$1 \rightarrow 2$

	0	1	2	3
0	0	1	1	0
1	0	0	1	1
2	0	0	0	1
3	0	0	0	0

4x4

Code -

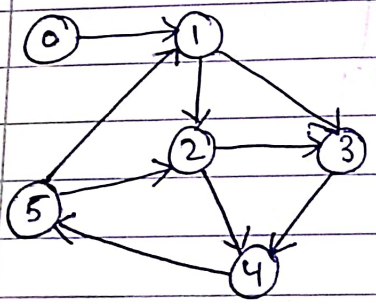
```
void createAdjMatrix(vector<pair<int,int>> edgelist)
{
    int n = edgelist.size();
    vector<vector<int>> adjMatrix(n, vector<int>(n,0));

    for(auto i : edgelist){
        int u = i.first;
        int v = i.second;
        adjMatrix[u][v] = 1;
    }
}
```

Time complexity - O(size of edgelist)

Space complexity - $O(n^2)$ where n is no. of nodes.

- **Adjacency list** - It's also a way to represent the graph. We will use map here in which key represents the node and value represents the neighbours corresponding to that node.

let a graph -



0 → {1}
1 → {2, 3}
2 → {3, 4}
3 → {4}
4 → {5}
5 → {2, 1}

This is how we represent a graph using adjacency list.

Code - directed / undirected graph -

```
class graph {
public:

 unordered_map<int, list<int> adjlist;

 void addEdge(int u, int v, bool direction){
    //1 → represents directed
    //0 → represents undirected
    if (direction == 1) {
       adjlist[u].push_back(v);
    }
    else {
       adjlist[u].push_back(v);
       adjlist[v].push_back(u);
    }
 }
};
```

Time complexity = $O(V+E)$ where V is no. of vertex/nodes and E is no. of edges

Space complexity = $O(V+E)$.

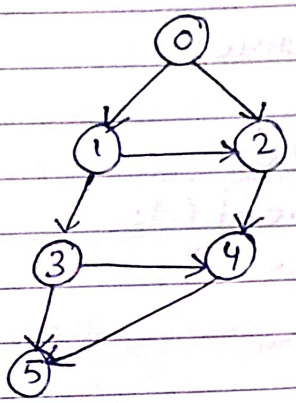
- Code – weighted graph

```
template <typename T>

class Graph{
public:

// T represents type of data stored by graph. In the
// map, the key represents node/vertex and the
// value represents a list of pairs. A pair
// comprises neighbour and distance/weight of that
// neighbour from a specific node

unordered_map<T, list<pair<T, int>>> adjlist;

void addEdge(T u, T v, int wt, bool direction)
{
        // directed -> 1
        // undirected -> 0
        if (direction == 1) {
            adjlist[u].push_back({v, wt});
        }
        else{
            adjlist[u].push_back({v, wt});
            adjlist[v].push_back({u, wt});
        }
}
};
```

* **Breadth-first Search Traversal (BFS traversal)**

Given is adjacency list for a graph, we have to do BFS traversal on that graph.

Let the graph be -



adjacency list

0 → {1, 2}
1 → {2, 3}
2 → {4}
3 → {4, 5}
4 → {5}

If any node in graph is already visited, we do not have to revisit it. for that, we will keep track of the visited nodes.

visited

0 → F T
1 → F T
2 → F T
3 → F T
4 → F T
5 → F T

~~0~~	~~1~~	~~2~~	~~3~~	~~4~~	~~5~~

= queue

output - 0 1 2 3 4 5

Logic -

To maintain initial state in queue, push the source node and mark it visited.

Now, pop out the front node from queue and check if neighbours exists for that front node. If yes, then push them out in queue. Repeat this until the queue is not empty.

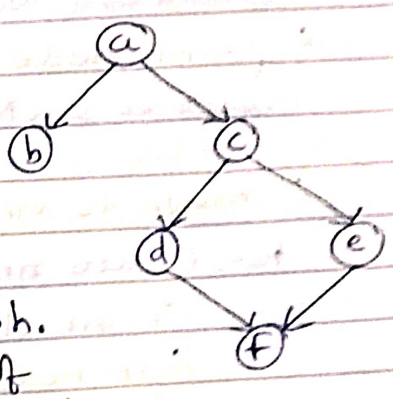
Code -

```
void bfsTraversal (T srcNode){

    unordered_map<T, bool> visited;
    queue<T> q;

    q.push(srcNode);
    visited[srcNode] = true;

    while (!q.empty()){
        T front = q.front();
        cout << front << " ";
        q.pop();

        for (auto nbr: adjlist[front]) {
            T nbrData = nbr.first;

            if (!visited[nbrData]) {
                q.push(nbrData);
                visited[nbrData] = true;
            }
        }
    }
}
```

* Depth - first search Traversal (DFS) -

Given is adjacency list of a graph, we have to do dfs traversal on graph.

let the graph be -



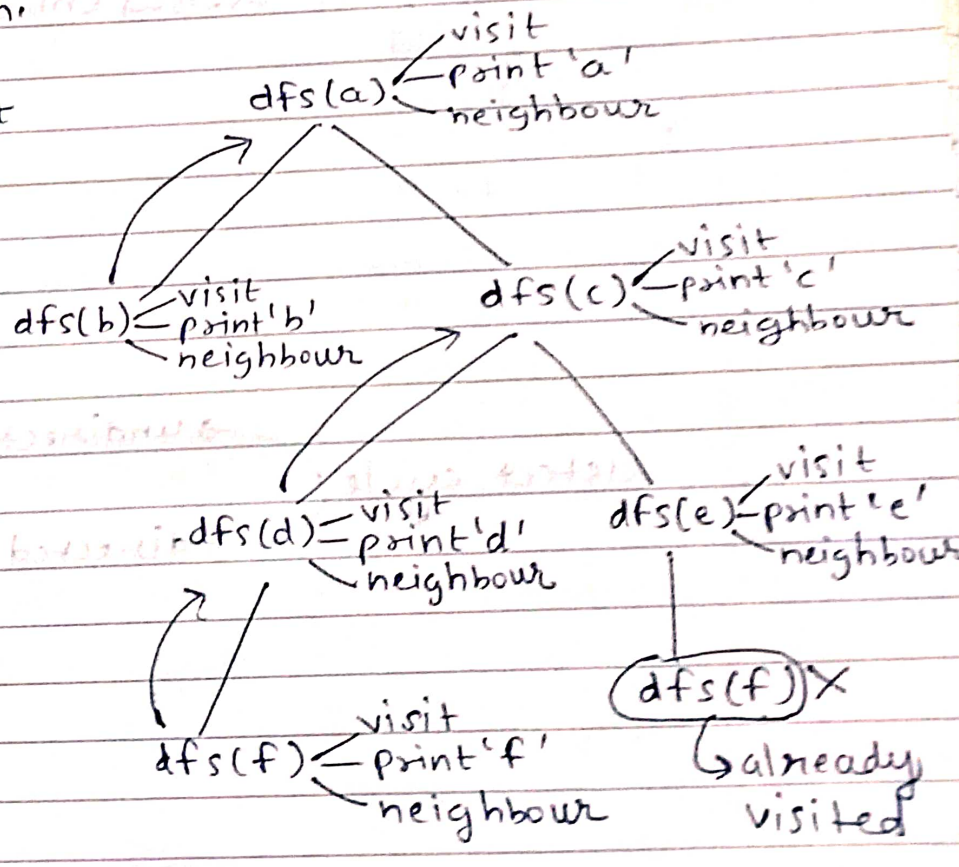
Using recursion, we can do dfs traversal on graph. We will go in one depth of graph. After returning, will go to the next depth.

adjacency list

a → {b, c}
b → { }
c → {d, e}
d → {f }
e → {f }

visited

a → ~~f~~ T
b → ~~f~~ T
c → ~~f~~ T
d → ~~f~~ T
e → f
f → ~~f~~ T



Logic - visit the node
Print it.
Go to the neighbouring nodes.

Code -

```
void dfsTraversal(T srcNode, unordered_map<T, bool> visited){

    //mark the node visited
    visited[srcNode] = true;
    // print node
    cout << srcNode << " ";

    // move to neighbours of node
    for(auto nbr : adjList[srcNode]){
        T nbrData = nbr.first;
        // if node is not visited, then go to it
        if(!visited[nbrData]){
            dfsTraversal(nbrData, visited);
        }
    }
}
```